Worksheet 4: *Datatypes*

Template file:	Worksheet4.hs
Labs:	Friday 10 and 17 March 2017
Hand-in:	21.00 hr on 19 Sunday March 2017
Topics:	Algebraic and recursive types. Trees, paths and errors.

1. A pack of playing cards contains 52 cards. Each card has a 'value' which is taken to be an element of the type

   ```
   data Value = A|Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|J|Q|K
   ```

   and it has a 'suite' which is taken to be an element of the type

   ```
   data Suite  = Hearts | Spades | Clubs | Diamonds
   ```

   A card is thus an element of

   ```
   type Card = (Value, Suite)
   ```

   (a) Define a show function for Value that transfers the values into the following strings: "A", "2", ..., "10", "J", "Q", "K",

   (b) Define a show function for Suite that transfers the elements of Suite into the strings: "H", "S", "C", "D"

   (c) (unassessed) Is it possible to write a show function for Card that would transform (A,Heart) into the string AH.

   (d) Give a concise definition of a value `pack::[Card]` containing all of the possible playing cards in some order. Use list comprehension.

   (e) There are two colours of playing card

   ```
   data Colour = Red | Black
   ```

   A card is Red if its suite is either Diamonds or Hearts and is Black otherwise. Write a function to determine the colour of a card.

   (f) A common way to shuffle a pack of cards is to repeatedly split the pack roughly in the middle and then to interleave the two portions. Write a function `split::Int->[a]-> Error ([a],[a])` so that `split n` divides a list in two at the point just after the `n`-th element (we start counting from `n=1`). For instance,

```
split 0 [1,2, 3,4,5] = Ok ([], [1,2,3,4,5])
split 2 [1,2, 3,4,5] = Ok ([1,2], [3,4,5])
```

Give an error in case the number is negative or larger than the length of the list. For instance,

```
split 8 [1,2, 3,4,5] = Fail
split (-5) [1,2, 3,4,5] = Fail
```

Write a second function to interleave two lists of type `[a]`, possibly of different lengths. For instance, interleaving two list of integers looks like this:

```
interleave [1,2,3] [4,5,6,7,8,9] = [1,4,2,5,3,6,7,8,9].
```

(g) A shuffle of the pack is specified by giving a list of integers. For example, the list `standard` below corresponds to the shuffle in which the pack is split after the 23rd card, interleaved, split again after the 26th card, interleaved, and so on.

```
standard :: [Int]
standard = [23,26,25,31,19,27]
```

Write a function `shuffle` which can be used on any list `xs` to calculate the effect of shuffling the list according to a list of integers.

Use the functions `split` and `interleave` defined in the previous part. Give an error in case `split` gives an error, i.e. give an error in case there is a number which is negative or larger than the length of the list `xs`.

2. A binary tree can be used as a database. Here, the leaves of a tree are either `ND` indicating *no data*, or `Data d` where `d` is a data item.

```
data Btree a = ND | Data a |  Branch (Btree a) (Btree a)
```

One can give a *path* to a leaf by giving a list such as `[L,R,L]` which indicates the leaf one arrives at by moving left, right, left from the root of the tree (of course, there may be no such leaf).

```
data Dir = L | R

type Path =  [Dir]
```

(a) Define `extract` which given a path and a binary tree, outputs the data at the end of the path, and gives an error value when the path does not match any data.

(b) Define `add`, whose three inputs are some data, a path, and a binary tree. The output consists of the binary tree, modified to include the data item at the end of the path. In more detail, if the input path ends in a leaf node of the form `ND` the tree is extended to contain the new data. If the path leads to a branching node or a leaf node of the form `Data d` an error value is given.

(c) Suppose the tree holds data of type `a`. Define the function `findpath`, which given a function `f`, some data `x` and a tree `t`, returns the lists of paths (possible empty) in `t` to the nodes of the form `Node d` where `f d` is equal to `x`.

3. Consider the following types:

```
data Tree a = U | F a (Tree a) (Tree a) deriving Show
```

```
type Person = String
```

We will use instances of this datatype to store genealogical information. Example: the information that Anna has Fer-Jan and Paula as parents will be denoted by

```
F "Anna" (F "Fer-Jan" U U) (F "Paula" U U).
```

It is convention that the second argument of `F` represents information about the father, and the last argument information about the mother. The letter `U` indicates the absence of information.

(a) Explain the phrase "deriving Show" in the declaration of `Tree`. Also explain why there is no need for such a phrase in the declaration of `Person`.

(b) What is the type of the function `putString`? Explain briefly the effect of the term `putStr "Hello"`.

(c) For later use in this question: write down your favourite parametrically polymorphic function

`sort :: Eq a => (a -> a -> Bool) -> [a] -> [a]`

that given a strict order relation `ord` and a given list `xs`, sorts `xs` using `ord`.

Example: `sort < [3,2,1] = [1,2,3]`

(d) In genealogy people occurring in a family tree are often labelled with a number, as follows: people at the root of the tree get number `1`. Next we apply the rule that, if people have number `n` then his/her father will get number `2*n` and his/her mother will get number `2*n+1`, until all people in the tree have been labelled.

Write down a function `genlabel :: Tree a -> Tree (Int,a)` that adds a number to the elements of a tree following the above practice.

Example: `genlabel (F "Fer-Jan" (F "Willem" U (F "Adriana" U U)) U) = F (1,"Fer-Jan") (F (2,"Willem") U (F (5,"Adriana") U U)) U` .

Larger terms in this F,U notation are pretty much unreadable. The family tree of Anna that includes her grandparents would look like:

```
F "Anna" (F "Fer-Jan" (F "Willem" U U) (F "Nettie" U U)) (F "Paula" (F "Mario" U U) (F "Martha" U U))
```

Genealogists have invented all sort of ways to print such terms either as lists as 2-dimensional trees.

(e) We want to print the above family tree of Anna as a list, using the genealogical labelling of the previous item:

```
1:  Anna
2:  Fer-Jan
3:  Paula
4:  Willem
6:  Mario
7:  Martha
9:  Adriana
```

Write a function `printlist :: Tree Person -> IO()` that takes a tree of type `Tree Person` and produces a list of labelled names, ordered by the genealogical label and printed as the above example, each label name of the form `n: name` on its own line.

Hint: given a tree, add genealogical labels to the tree, flatten the tree to a list of pairs of type `(Int,Person)` and then pretty print the list as indicated by the above example: each pair on its own line.

To finish, we want to print the above family tree of Anna as a 2D-tree as follows:

```
                Willem
        Fer-Jan
                Nettie
Anna
                Mario
        Paula
                Martha
```

In this format the information of a person is printed between the lines of the information of his/her parents, in so far as there is information of them given. The indentation is produced with tab characters. If a person has `k` generations distance from the root of the tree, then we use `k` tab characters for indentation.

The above tree was the result of printing the following string:

```
"\t\tWillem\n\tFer-Jan\n\t\tNettie\nAnna\n\t\tMario\n\tPaula\n\t\tMartha\n\n"
```

(f) Write a function `print2Dtree :: Tree Person -> IO()` that takes a tree of type `Tree Person` and produces a family tree as above.

Hint: one possibility is to proceed more or less as in the previous item. But take care that you print the names in the proper order: children between their parents. Change the pretty printing, so that the pairs pretty-print as indicated by the above example: each pair `(n, person)` will be decoded in a string of characters beginning with a suitable number of tab characters.